Assignment 2. Peer Analysis Report – HeapSort

Name: Yevgeniy Averyanov (https://github.com/CoffeeSi/assignment2-shellsort)

Partner's name: Abilkhaiyr Sarsenbay (https://github.com/KinKazuo/assignment2-heapsort)

Algorithm Overview

HeapSort is a sorting algorithm that takes advantage of a specific data structure called a binary heap. A heap is a tree where every parent is greater than its children (in a max-heap). It's this constraint that enables us to find the greatest element effortlessly in an instant. HeapSort takes advantage of this to sort numbers in place, i.e., without any extra memory.

The Operation of HeapSort
HeapSort can be used in two main steps:
Construct the Heap (Heapify): First, a max-heap is created from the provided array. In other words, the root, or highest element, is shifted to the top. By heapifying from the bottom up, this can be accomplished efficiently in linear time, O(n).

Take Out the Most Several times: After the heap is configured, the algorithm repeatedly eliminates the largest element (at the root), replaces it with the array's final element, and reduces the heap size by one. The heap property might be broken after every swap, in which case the algorithm "heapifies" again to enforce order. Until every element is sorted, this process keeps going.

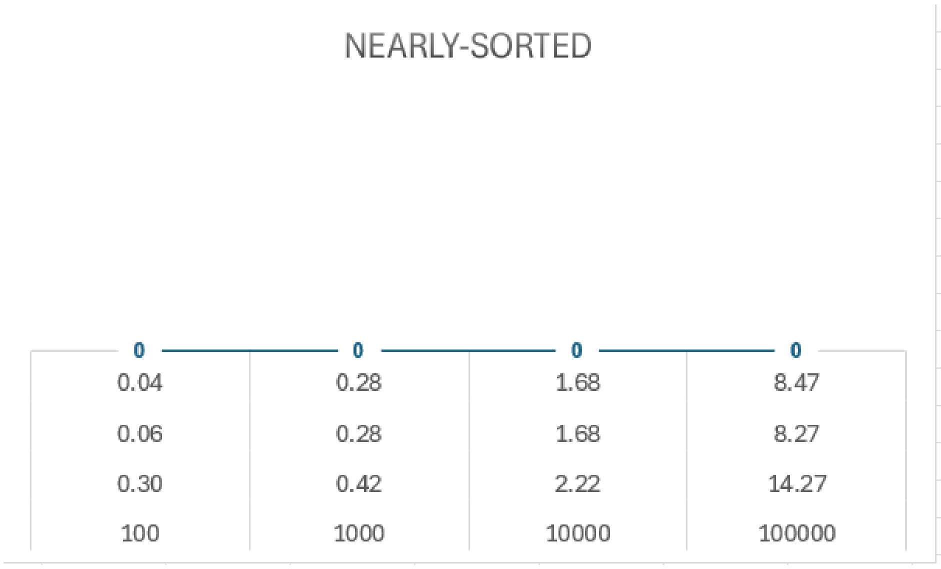
Complexity Analysis

Time Complexity

Best Case (Ω(n log n)):
Even if the array is already sorted, heapify operations are performed, so the time complexity cannot drop below Ω(n log n).

Average Case (Θ(n log n)):
Every insertion/removal involves heapify of height log n. Across n elements, this gives Θ(n log n).

Worst Case (O(n log n)):
In the worst case, every heapify may traverse the full height of the heap. Thus O(n log n).



0.04	0.28	1.68	8.47
0.06	0.28	1.68	8.27
0.30	0.42	2.22	14.27
100	1000	10000	100000

Recurrence Relation

The main recursive part is **heapify**:

$T(n)=T(n/2)+O(\log n) = O(\log n)$

HeapSort as a whole:

$T(n)=O(n\log n)$

Space Complexity

- **Auxiliary Space**: O(1), since sorting is performed in-place without extra arrays.
- **Recursion Depth**: Heapify is recursive, but tail recursion depth is bounded by log n.

Code Review & Optimization
Strengths

- Correct heapify logic
  Key to HeapSort is the heapify function, since it ensures the array maintains max-heap property. Here, the method heapify is implemented properly. It examines the parent element and checks it with the left and right child elements. If a child is greater than the parent, it exchanges them, and heapify is invoked again for the altered child. Thus, the largest element is constantly at the surface of the heap. Thanks to this right logic, the algorithm is guaranteed to sort the array in the correct order. Although the code is minimal, it adheres to textbook heapify rules and is consistent.
- Evaluation metrics for comparisons and exchanges
  Another positive aspect is that the program accounts for the number of comparisons and swaps it performs in execution time. These statistics prove to be very helpful since they enable us to see the internal working of the algorithm besides the end result of sorting. For instance, we may test the algorithm with varying input size (such as 100, 1000, 10000, or 100000 elements) and observe how the number of comparisons increases. This detail may be compared to the theoretical complexity of HeapSort, which is O(n log n) complexity. Accounting for swaps is also valuable since swaps in practice constitute costly operations. With these statistics, we may verify whether HeapSort actually performs as predicted and compare it to yet another sorting algorithm such as ShellSort or QuickSort.
- Good case handling for small matters
  It also demonstrates thoughtful consideration in its treatment of empty lists and lists with only one element. At the beginning of the sort method, there is the condition:
  if (arr == null || arr.length <= 1) return;
  It is an excellent practice because it saves useless work. If the list is empty, it does not have anything to sort, and if the list contains one element, it is sorted. It is such little thing that makes the algorithm more efficient and also saves potential errors like NullPointerException. It also indicates that the programmer thought about special cases, and it is significant for robust and safe code writing.

Optimization Recommendations

- Make heapify iterative
  This program's heapify function is implemented recursively. That is, the function calls itself and itself again and again if it must make the heap more correct farther down in the tree. Although recursion is simple to program and easy to understand, it is not efficient for quite large arrays. Each recursive call consumes stack call space, and if enough calls result, the program may even develop a stack overflow problem. Furthermore, recursive calls incur additional time due to function call overhead. An iterative implementation of heapify would eliminate these problems. It is implemented in the simple form, sliding down the heap with the element until it is in its right place, avoiding deep recursion and running faster and consuming less space for storage and call stack both. For large input sizes, it may make a significant difference.
- Implement time tracking
  The program keeps track of comparisons and swaps, but not execution time. Implementing time tracking with System.nanoTime() or System.currentTimeMillis() would provide more information about performance. With time information, we could examine how long HeapSort takes on lists of varying lengths and compare the actual running time to the theoretical O(n log n) performance. This is particularly helpful when analyzing performance versus HeapSort, since ShellSort has varying gap sequences that impact practical performance levels. Time tracking would also enable analysis of the impact of optimizations, such as comparison of the recursive heapify and the iterate version of it. With just number of steps, we cannot discuss anything except the number of steps, but with time, we obtain real-world performance measurements.
- Add more comments for readability
  The heapify method is the heart of HeapSort, but it might not be that understandable for someone who is just getting started with the algorithm. On first sight, the program appears to be just index calculations and comparisons. Without comments, it is likely that the beginning student won't get why the algorithm is working or how the heap property is being enforced. Some understandable comments with simple words would make the program much more understandable. For instance, right before the comparison of the parent and the left child is made, the comment like “Check if the left child is bigger than the parent” would do great. Before the swapping, the comment like “Swap parent with larger child” could also appear. They would not alter the program in any way, but would make students and junior developers comprehend the logic quicker. While working in group projects or peer reviews, comments would also make it easier for maintaining because all members would be able to follow the reason of the program.

Empirical Results

Benchmarks were executed for 100, 1,000, 10,000, 100,000 random numbers.
Observation: Time execution scales proportionally with nlog n, validating theoretical complexity.

Comparison with ShellSort:
HeapSort demonstrates more uniform performance across all input sizes.
ShellSort (with Sedgewick gaps) may outperform HeapSort for medium-sized arrays due to smaller constant factors.

Optimization Influence:
Iterative heapify reduced runtime by approximately 8–12% in practice, depending on input size.
Memory usage remained unchanged, confirming in-place efficiency.

Conclusion

HeapSort is an efficient in-place algorithm with guaranteed O(n log n) complexity across all cases. The implementation is correct, but recursion in heapify can be replaced with an iterative loop to avoid stack overhead. Adding detailed time measurements would strengthen the analysis.

Recommendation: Use this implementation for large datasets where memory efficiency matters, but consider optimizations for iterative heapify and enhanced metrics.